$2.50

Popular Computing

The magazine for those interested in the art of computing

March 1978 Volume 6 Number 3

90	77	101	72	96	16	135	133	76	60	62	84
17	75	38	13	56	82	29	104	61	113	88	57
92	68	97	100	118	95	119	120	30	106	18	127
10	35	73	51	50	66	144	140	129	22	78	125
122	117	14	2	3	109	39	20	9	5	115	58
139	4	91	105	71	128	85	36	8	65	136	54
41	111	40	89	27	19	15	59	26	43	116	63
70	138	31	94	55	69	52	86	34	21	132	11
6	24	102	103	108	47	93	134	114	80	126	12
81	87	121	112	99	79	33	124	48	28	107	123
74	67	131	83	137	141	25	44	23	37	7	110
143	98	49	46	130	1	45	64	53	142	32	42

A NEW PROBLEM...and our 16th Contest!

Reversals-By-Fours

We have an interesting new problem and a new contest--our 16th.

Let us describe exactly how the pattern on the cover was constructed. We started with the 12 x 12 array filled with the numbers from 1 to 144 in normal order (that is, 1 to 12 across the top row; 13 to 24 across the second row; ... 133 to 144 across the bottom row).

We now selected four adjacent cells (either horizontally or vertically) and reversed their order. For example, the four adjacent cells starting with the 5th cell in the top row are:

5	6	7	8

and the reversal produces:

8	7	6	5

Every set of four cells in the 12 x 12 array can be identified according to the pattern shown in Figure A. There are 216 such sets, and each is identified by its top (if vertical) or leftmost (if horizontal) element. Thus, the set used in the example above is number 113 in the pattern of Figure A.

To illustrate the reversing procedure further, pattern B shows the effect of applying the process to sets 1, 2, 3, 4, 5, 6, 7, 8, and 9 in order, starting with the initial contents of the first column of the array.

So here we are. Starting with the initial ordering in the array, we selected 1000 sets of four adjacent cells at random (by generating random integers in the range from 1 to 216 and applying the result to pattern A) and reversed them. Pattern C shows precisely the 1000 selections that were made. The pattern on the cover is the end result of the 1000 reversals.

Continued on page 5.....

* *

Publisher: Audrey Gruenberger
Editor: Fred Gruenberger
Associate Editors: David Babcock
Irwin Greenwald
Contributing Editors: Richard Andree
William C. McGee
Thomas R. Parkin
Advertising Manager: Ken W. Sim
Art Director: John G. Scott
Business Manager: Ben Moore

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each. Copyright 1978 by POPULAR COMPUTING.

@ 2023 This work is licensed under CC BY-NC-SA 4.0

A

001 109	010 110	019 111	028 112	037 113	046 114	055 115	064 116	073 117	082	091	100
002 118	011 119	020 120	029 121	038 122	047 123	056 124	065 125	074 126	083	092	101
003 127	012 128	021 129	030 130	039 131	048 132	057 133	066 134	075 135	084	093	102
004 136	013 137	022 138	031 139	040 140	049 141	058 142	067 143	076 144	085	094	103
005 145	014 146	023 147	032 148	041 149	050 150	059 151	068 152	077 153	086	095	104
006 154	015 155	024 156	033 157	042 158	051 159	060 160	069 161	078 162	087	096	105
007 163	016 164	025 165	034 166	043 167	052 168	061 169	070 170	079 171	088	097	106
008 172	017 173	026 174	035 175	044 176	053 177	062 178	071 179	080 180	089	098	107
009 181	018 182	027 183	036 184	045 185	054 186	063 187	072 188	081 189	090	099	108
190	191	192	193	194	195	196	197	198			
199	200	201	202	203	204	205	206	207			
208	209	210	211	212	213	214	215	216			

B

001	037	037	037	037	037	037	037	037	037
013	025	049	049	049	049	049	049	049	049
025	013	001	061	061	061	061	061	061	061
037	001	013	025	073	073	073	073	073	073
049	049	025	013	001	085	085	085	085	085
061	061	061	001	013	025	097	097	097	097
073	073	073	073	025	013	001	109	109	109
085	085	085	085	085	001	013	025	121	121
097	097	097	097	097	097	025	013	001	133
109	109	109	109	109	109	109	001	013	025
121	121	121	121	121	121	121	121	025	013
133	133	133	133	133	133	133	133	133	001

155	124	7	169	17	6	141	63	106	181	100	60	23	107	95	129	207	125	178	120
103	51	65	142	71	77	167	132	143	216	190	11	187	136	47	143	181	150	94	32
188	37	41	150	44	123	8	99	64	6	81	135	194	2	86	133	211	38	54	12
119	4	70	205	109	167	185	157	97	51	175	154	150	1	39	111	75	144	157	205
153	135	42	21	190	208	34	201	172	13	100	53	134	84	79	61	139	37	96	159
159	79	75	145	80	133	81	187	70	146	15	42	29	190	166	134	64	57	58	66
79	186	126	100	204	151	201	96	139	36	210	158	26	44	77	52	35	206	147	147
3	82	188	77	185	100	16	29	133	108	32	211	28	23	165	167	154	97	41	190
47	57	191	8	132	65	187	38	39	136	213	145	87	107	33	140	111	46	119	186
196	143	204	156	79	214	28	9	45	11	114	12	92	30	26	61	181	130	137	168
85	195	34	75	124	138	36	67	20	29	25	69	183	36	114	185	47	79	169	198
177	180	121	106	147	133	19	81	6	105	32	190	131	156	6	216	154	74	86	135
202	83	37	67	153	181	107	51	147	162	173	180	6	9	29	158	163	149	145	110
1	157	133	185	106	148	92	154	23	13	118	197	159	64	128	201	80	198	36	56
127	70	52	6	92	179	67	82	164	115	51	38	41	114	210	135	122	11	30	78
33	33	117	117	115	79	158	120	67	13	129	185	138	202	30	111	153	57	83	13
36	99	133	32	134	107	192	138	43	163	203	130	73	193	118	196	157	53	88	187
131	72	122	108	82	20	107	20	143	72	159	159	159	190	95	173	208	29	108	199
142	163	191	133	83	124	191	55	104	204	98	13	163	200	175	203	46	123	44	44
211	66	78	40	202	119	51	179	151	124	124	70	98	147	195	191	43	165	99	130
4	74	141	136	191	16	140	128	131	183	112	79	91	113	153	59	190	78	42	167
104	45	6	145	159	111	164	200	26	46	74	37	113	127	7	192	169	23	197	98
176	76	116	207	201	20	66	4	193	112	21	111	62	170	169	83	167	179	13	114
157	129	77	187	200	159	30	139	164	162	94	1	123	166	191	203	145	200	24	135
133	98	168	61	59	113	14	110	53	198	208	66	59	123	23	147	17	18	125	7
63	48	70	72	1	109	158	23	102	197	189	85	131	175	22	140	159	26	83	148
54	173	32	161	145	164	214	155	160	31	211	124	6	44	215	28	145	151	169	26
179	200	31	156	72	104	208	117	34	162	118	31	63	52	183	38	108	167	42	39
128	144	3	25	165	187	29	147	117	46	62	104	195	174	53	25	37	122	69	55
37	14	137	158	65	151	178	168	215	42	149	81	47	133	86	188	102	181	158	191
214	138	6	119	112	81	64	39	92	86	31	88	126	42	6	193	132	84	82	127
14	126	52	152	26	42	192	85	63	53	13	51	176	63	71	117	33	94	44	134
65	117	100	215	76	161	85	145	210	51	78	106	60	103	86	146	155	177	58	129
112	146	146	177	49	47	13	109	210	164	143	53	200	170	72	76	133	171	149	153
120	194	34	47	188	83	109	194	75	33	81	161	131	43	93	74	182	39	133	79
168	20	35	208	73	30	186	35	142	194	35	56	202	164	147	79	158	173	148	19
54	13	16	97	41	78	101	21	153	147	204	190	12	37	211	171	106	183	39	101
204	20	74	36	24	200	86	80	93	40	188	10	127	140	60	145	12	40	47	214
57	203	113	143	205	149	37	180	54	80	102	143	21	192	12	191	105	51	67	112
41	212	163	151	94	54	201	159	34	37	85	112	83	196	80	40	58	73	129	68
86	88	205	104	194	60	89	100	191	54	206	208	194	132	183	11	168	91	3	67
132	64	1	169	73	56	10	166	103	166	193	116	136	190	188	6	64	155	177	121
118	55	89	150	24	101	92	48	112	204	122	88	152	1	75	106	179	86	141	125
155	133	106	201	120	200	114	160	89	33	208	92	24	59	64	120	10	200	123	94
102	179	138	87	190	51	1	204	207	98	178	137	169	106	133	2	150	200	57	91
112	215	178	199	141	10	165	54	126	72	31	1	81	68	74	5	161	84	184	13
54	60	158	177	31	212	59	85	105	103	96	47	12	31	145	108	140	8	110	69
58	3	173	76	187	185	178	74	17	75	12	74	138	98	21	183	200	140	56	175
61	88	105	136	137	209	14	49	213	96	167	37	67	59	213	1	108	33	55	169
176	101	15	195	102	63	37	66	198	7	36	16	75	165	180	110	6	68	183	165

What we seek is a method of attack on this problem. We know how to go from the cover pattern back to the initial state--just work this sequence backwards--but how would you go about it for the next case?

C

PROBLEM 224

The problem we have is to take the cover pattern, reverse sets of four adjacent cells, and return the array to its initial ordering. If the 1000 reversals dictated by pattern C were executed in reverse order, the job would surely be done. Just as surely, though, there is some smaller set of reversals-by-four that will do the same job. If a set of four is reversed twice, the net effect is nill. We observe such futile actions on lines 7, 16, 18, 19, 20, and 34 of pattern C.

Our 16th Contest Problem then is this: produce a list of reversals that will restore the array to its initial ordering. For the person who does the best job (but not necessarily the shortest list) by computer, in the opinion of our panel of judges, a prize of a TI-58 programmable calculator will be given.

All entries must be received by June 15, 1978 at

Contest 16
POPULAR COMPUTING
Box 272
Calabasas, California 91302

* * * * *

GLOSSARY

BOTTOM UP TURKEY n. (fr. the Greek turkey, meaning yahoo). One who believes that COBOL programming begins at the keypunch.

TOP DOWN CLOWN One who believes devoutly that a canonical scheme for organizing a problem solution will clear up his muddled thinking.

FLOWCHARTING A scheme for organizing the logic of a problem solution that is obsolete, archaic, and disapproved, and for which no suitable substitute has been found.

KLUDGE An ill-assorted collection of poorly-matching parts, forming a distressing whole. (J. Granholm)

STRUCTURED PROGRAMMING A systematic collection of techniques, procedures, and low cunning in the process of coding that have been used by all master coders since time began.

HOTSHOT A demon coder who can write any given I/O routine in ten minutes, but who takes two years to get it working in the system.

CE "Customer engineer." A person who has been trained to convince users of computers that their troubles are imaginary or, at best, the result of unusual atmospheric conditions.

BOOK REVIEW

Standard FORTRAN Programming   Third edition

By Donald H. Ford
Richard D. Irwin, Inc., 1978.
Soft cover, 6 x 9, 330 pages, $8.95.

Authors have one big problem when it comes to reviewers of their book. Reviewers frequently review the book that they wish had been written--and then always wind up castigating <u>this</u> book for not measuring up. It is patently unfair, but it is an easy trap for any reviewer to fall into. So let's be sure to review the book that Ford wrote, and not the one we might prefer him to have written.

He wrote <u>Basic FORTRAN IV Programming</u> in 1971 and revised it in 1974 to include optional features of the language. This new book (which is basically a third edition) restricts itself completely to ANSI standard Fortran. In any event, it is intended for a first course in programming (which means mostly coding). One would expect a third edition of a textbook to be thoroughly debugged, tested, and well put together (nine out of ten published texts do not sell out their first printing).

If the textbook is on programming, then one would expect the author to appreciate the distinction between debugging and testing of programs. Ford distinguishes between the bugs that can be detected during compilation and execution (that is, the mechanical bugs that can be detected by the compiler or by the subroutines) from the logical errors in the program, but he offers little help for curing the latter type.

The book claims to be a useful adjunct to the learning of Fortran programming. Some of its pedagogy is questionable. For example, showing beginners coding that includes:

```
SQRT(A**4.0)
```

will encourage them to think in just such a sloppy way (doing a little algebra wouldn't hurt, and if the fourth power is really intended, then the use of 4.0 instead of 4 will force Fortran into a logarithm calculation, which won't improve the solution of the problem). Similarly, the student ought to be told that

```
ALØG(A*A)
```

can be simplified.

Or, consider this example: control is to pass to statements 55, 65, or 75 according as the contents of KØDE is 6, 7, or 8. The following statements will do it and edit the data at the same time:

```
          IF (KØDE - 6) 3, 98, 99
98        IF (KØDE - 8) 99, 99, 3
99        KKØDE = KØDE - 5
          GØTØ (55, 65, 75) KKØDE
```

(statement 3 is a STØP)--but what horrible computing this is! Students will discover soon enough how to make their programs incomprehensible without being encouraged by their textbook. The inclusion of COMMENTS to explain away such cutesy coding will not clear it up. If you want to edit KØDE to certify its value as lying within the proper range, then for heaven's sake do so.

The author states in the Preface that he "purposely stays away from 'high-powered' mathematics." Such disclaimers in textbooks usually indicate that the author doesn't know any high-powered mathematics; if he did, he would surely include some of it, in starred exercises, or footnotes. Fortran is, after all, a scientific and engineering language, and the built-in functions like SIN and LØG are there to be used.

Ford emphasizes that this third edition conforms to the ANSI standard for Fortran. It does not conform to the ANSI standard for flowcharting. For that matter, flowcharts are used very little in the text, and then only for trivial situations.

The approach to a sorting program, while logical and clearly stated, again fosters the worst kind of computing (namely, inefficient, complicated, and difficult to maintain or modify).

The list of things that are just slightly off in this text could go on and on. The point is that the book could be tremendously improved with a few changes--and should have been by the time it went to a third edition.

But in spite of all this, and in comparison to competitive books, and considering the price, the Ford text is not a bad choice for a beginning class. The subject matter is in a logical order; the exposition is clear and concise; the approach is probably pleasing to students; and the typography and layout are excellent. □

BUBBLE SORTING...AGAIN

In issue 58, in the article on Shell sorting, a flowchart for interchange (bubble) sorting was presented.

In bubble sorting in ascending order, each element of the set to be sorted is compared to its immediate neighbor, and the two elements are interchanged if necessary. The entire set is swept from left to right repeatedly until a sweep is performed that involves no interchanges. Thus, the last sweep is always only a sequence check.

The first sweep surely moves the largest element of the set to the far right end. Thus, as Dr. Neal Koss (Yale University) points out, if each subsequent sweep examines every element, the process loses efficiency. He offers the flowchart shown on the facing page as an improvement; the number of elements to be examined in each sweep of the data is reduced by one.

The saving in machine time for Dr. Koss's scheme is a function of the number of items being sorted as well as the amount of ordering that already exists in the data (for example, for data already in sequence, the new scheme will run slower). As the block of data is shortened, the only saving is the comparison at Reference 3 (there cannot be any interchanges saved), and the price to be paid is the extra test and modification indicated by the heavy bracket.

There are many ways in which bubble sorting can be improved. The sweeps can alternate directions ("Double Bubble"), and the block can then be shortened at both ends. Or, the block can be shortened even faster by keeping track of the addresses of the last interchange. With all possible speedups included, one might think of it as almost a new way to sort.

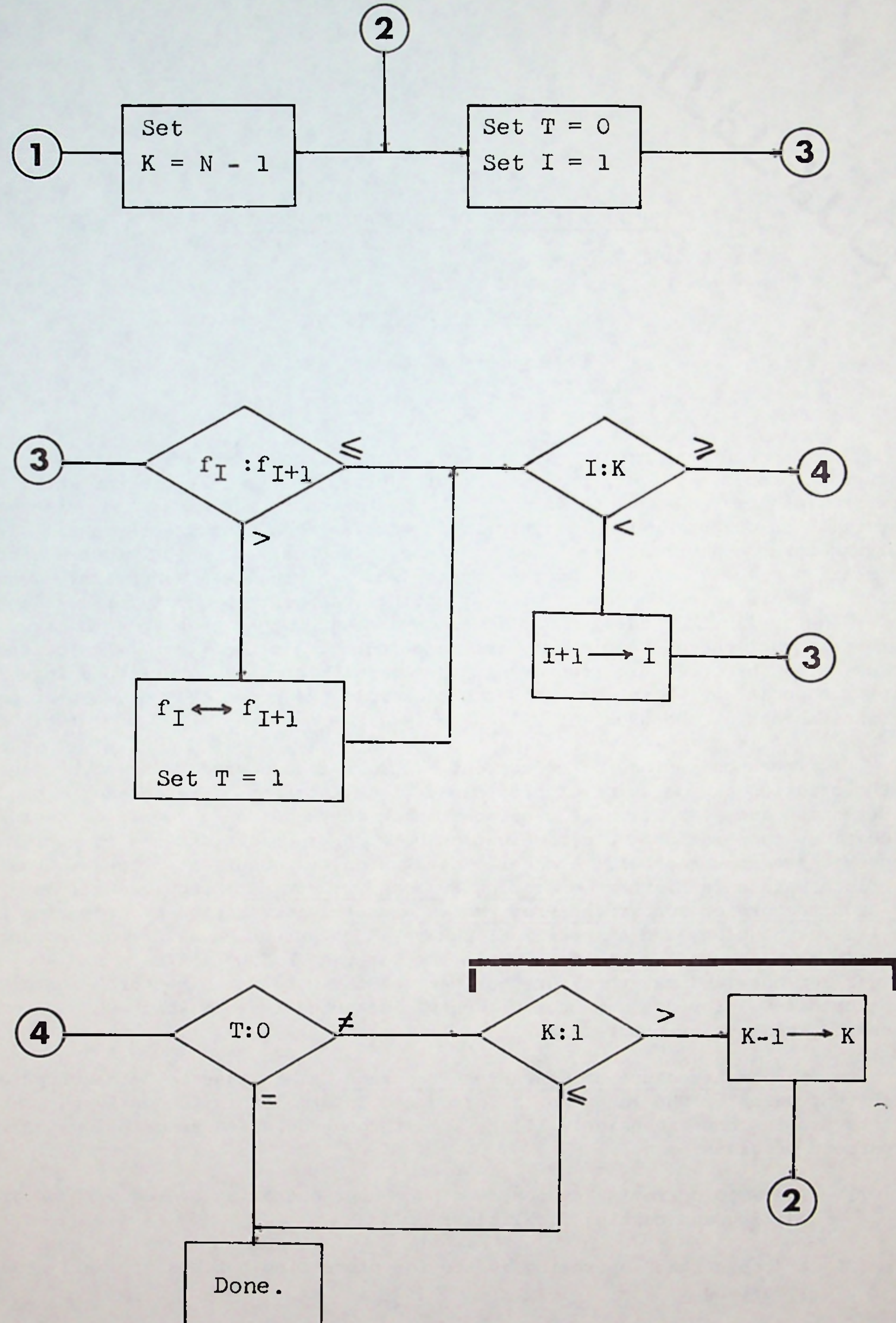
There is really a question of computing philosophy here. Bubble sorting is intrinsically inefficient. Its virtue is its simplicity; that is, the ease with which it can be quickly coded in any language, when a sort is needed in a hurry under the proper conditions. What are those conditions? Some combination of:

(a) A few things (say, 15 or less) to be sorted many times; or

(b) Many things (say, 100) to be sorted a few times; or

(c) Any sorting situation in which speed of production of the program outweighs other considerations; that is, where elapsed time to results is the paramount criterion.

In other words, techniques like bubble sorting are inefficient, but are useful when conditions warrant.


1
Set
K = N - 1
2
Set T = 0
Set I = 1
3
3
f_I : f_I+1
≤
>
f_I ↔ f_I+1
Set T = 1
I:K
≥
4
<
I+1 → I
3
4
T:0
≠
=
K:1
>
K-1 → K
≤
2
Done.

BOOK REVIEW

THE ARCHITECTURE OF CONCURRENT PROGRAMS

by

Per Brinch Hansen

Reviewed by W. C. Mc Gee

A concurrent program is one which controls the execution of two or more concurrent processes, i.e., processes that overlap in time. A program which reads data from an input device, performs some computation on the data, and writes the results to an output device is typically organized as a concurrent program with at least three concurrent processes: an input process, an output process, and a computation process. A time-sharing system which time-slices a computer among a number of users is a concurrent program having a separate process for each active user. In the first of these examples the processes cooperate with each other, whereas in the second example they are independent. In the first example, the processes may be truly simultaneous if the computer provides overlapped input/output/computation, whereas in the second example the processes only appear to be simultaneous at the user's level.

The architecture of concurrent programs is concerned with such questions as the creation and deletion of processes, the allocation of resources to processes, to detection and resolution of deadlocks which may occur as a result of resource allocation, cooperation and communication between processes and the isolation of processes from one another. A general discussion of concurrent processes was given by Brinch Hansen in his text Operating Systems Principles (Prentice Hall 1973). The Architecture of Concurrent Programs extends this discussion by proposing a specific methodology and a specific language - Concurrent Pascal - for designing and writing concurrent programs. While the later book does not have the scope of the earlier one (and may therefore not warrant the word "architecture" in the title), it nevertheless provides a number of useful ideas and lessons which should not be missed by students of programming.

The Architecture of Concurrent Programs is, in essence, a description of Concurrent Pascal. The language is introduced informally in the early chapters, and its use is then extensively illustrated through detailed descriptions of three concurrent programs:

(1) The SOLO operating system, a single-user operating system with program editing and testing facilities;

(2) A "job stream" operating system for processing batches of small student jobs; and

(3) A real-time scheduler for process control applications.

A more formal description of the language and its implementation is given in the final chapters.

Concurrent Pascal was developed by the author between 1972 and 1975 while at Cal Tech. It is an extension of the sequential programming language Pascal, invented by Niklaus Wirth. Sequential Pascal is an ALGOL-like language whose most distinctive feature is its provision for user declaration of data types. Through such declarations, the PASCAL compiler is able to detect many programming errors that would otherwise not be apparent until the program is executed, when they are much harder to isolate and correct.

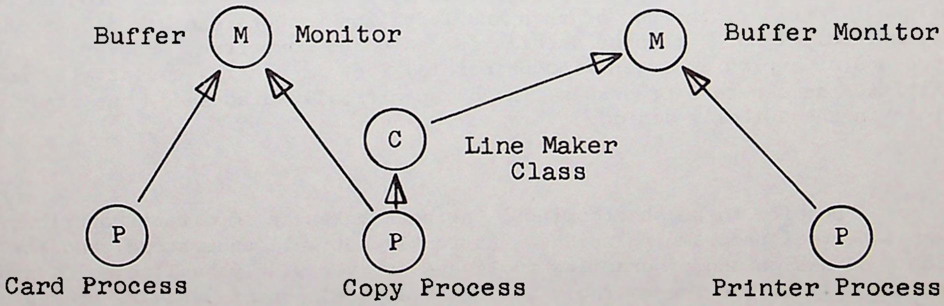
Concurrent Pascal extends Pascal by adding three concepts: the <u>process</u>, the <u>class</u>, and the <u>monitor</u>.

A <u>process</u> is a sequential procedure having its own private data structure. One or more processes are initiated at the start of program execution, and thereafter run in parallel until they stop themselves or until the host system is shut down (a "do forever" statement permits processes to cycle continuously).

A <u>class</u> is a data structure and a set of procedures which may be invoked by a single owning process for purposes of accessing the class' data structure. Classes facilitate top-down program design and enhance program reliability by shielding data structure details from processes and by providing formal, structured access to data structures. The class is similar to the <u>abstract data type</u> notion currently in vogue.

A <u>monitor</u> is structurally the same as a class, except that its procedures may be invoked by any process - it is, in fact, the method by which processes communicate. Provision is made for only one process at a time to be active within a monitor, thus preserving the integrity of the monitor's data structure. This data structure may contain one or more single-position <u>queues</u> for achieving inter-process synchronization: a process places its identifier in a queue while waiting for an arbitrary prescribed condition to become true; when another process sets the condition "True", the first process is automatically resumed.

Depicted below is an example of a concurrent program which reads and prints cards. The arrows denote access rights which are declared by the programmer to preclude unintentional access.



Concurrent Pascal has been implemented by the author and his colleagues on the PDP 11/45. The implementation consists of a compiler which generates code segments for a virtual machine, an interpreter which executes the virtual machine code, and a "kernel" which uses a time-slicing approach to allocate the computer to concurrent processes.

Concurrent Pascal inherits from Pascal the advantages of structured programming and extensive compile-time checking. The author's experiences with the language are truly impressive:

- The SOLO operating system required less than two person years to develop, whereas with conventional assembler coding the same task would have taken 20-30 person years (author's estimate)

- The job stream operating system was designed, coded, and tested by the author in ten (10) days.

- The original version of the real-time scheduler, written in assembler language, took one-half year to develop; in concurrent Pascal it was done in 3 days.

A limitation of Concurrent Pascal is its lack of provision for creating and deleting processes during program execution. This lack is of little moment when the number of processes that can be profitably employed tends to remain constant, as is the case in each of the author's example programs. When this number varies, however, Concurrent Pascal forces the programmer to anticipate the largest number of processes that will be required during program execution, and to dedicate resources to these processes even when they are not being used. A case in point is the on-line query system, in which a separate process is typically associated with each active user. As the number of active users undergoes its characteristic daily variation, it would be advantageous to allow the number of processes to follow suit.

A major shortcoming of the book is that it provides little insight into the matter of postulating a process configuration for a concurrent program. Since the same external program behavior can be achieved with many different process configurations, a discussion of the criteria available for choosing among alternative configurations would seem to warrant a central position in a book such as this one. In general, the author's choices of process configurations are motivated by performance considerations, in particular, the goal of keeping peripheral devices running at full speed and of overlapping peripheral device and central computer operation as fully as possible. To do this successfully, however, he requires detailed knowledge of the performance not only of peripheral devices but also of the program being designed! Further, the use of such knowledge runs counter to his dictum that we "deliberately ignore" machine details in designing concurrent programs. The use of processes as a program organizing principle, i.e., rendering essentially independent activities as seperate processes, is not specifically discussed, but clearly plays a role in the author's design.

Despite these shortcomings, the book makes an important contribution to an aspect of programming which is currently not well understood, and for which good design and programming tools are not yet widely available. Concurrent programs will become increasingly important as the cost of hardware processors continues to decline and multi-processor systems become the rule rather than the exception. ☐

how the computer affects you

6.1 THE INTERFACE

It is time to discuss more fully the ways in which computers are used in the service of society, and the ways they might be used in the future. Probably the most important point to make at the outset is the deep commitment to the use of computers that already exists. There is no turning back: modern society is already highly dependent upon computer control of countless activities and this involvement can only increase. There is simply no point in considering any present day version of the Luddites uprising wherein workmen in England, around 1812, tried to prevent the use of labor-saving machinery by attacking the machines and factories.

In other words, we are stuck with computers whether we like them or not. That is *not* to say that we must meekly accept any computer application that anyone invents, if it conflicts with other recognized goals of society. Of course, the question then is, recognized by whom? What may be regarded as a vicious use of computers by one group may be considered a beneficial (e.g., profitable) use by some other group. Since our society functions by checks and balances, the consensus of the majority tends to prevail. There are socially acceptable ways to influence courses of action, such as legislation and consumer pressure.

Other ways to influence systems are suggested from time to time. It has been openly recommended that people deliberately "fold, mutilate, and spindle" any punched cards whose use is regarded as offensive. Another such idea is that of feeding computerized systems false information. Any such schemes can have, at best, limited and local effects, and may even boomerang.

Consider the recommendation to damage a punched card that is designed to go back into a computerized system. The card relates to you; who will be hurt the most if the damaged card does any damage to the system? Moreover, if the system was designed with any intelligence at all, it took into account the fact that punched cards that are put into the hands of people who are not familiar with them may return in less than perfect condition. One precaution, for example, is to make two identical cards, and send one to the customer. On return, the customer's card is matched up with its mate, and the latter is fed back into the system.

The main point is that computerized systems are usually designed to serve their owners (and only incidentally to serve the public) and any effort by the public to upset the system can be readily countered.

6.2 THE BRIGHT SIDE

Let us now consider some of the applications of computers that appear to benefit us in one way or another.

Probably the largest single application of computers is for mechanized paper handling. This includes billing procedures (e.g., oil company or department store bills); accounts receivable and accounts payable; credit checking; subscriptions or book club accounting; reservations systems (e.g., airline ticketing, car rental, and hotel room reservations); and bank and insurance paper shuffling.

When these things function properly, both the customers and the owners of the systems involved benefit greatly. The benefits include:

1. Lowered costs. In every one of these systems, there is a large volume of similar data to be processed. Once the procedure has been programmed, the processing per item can become, and does become, very inexpensive.

Consider, for example, the processing of a personal check at a bank. If everything is normal, each check goes through 19 stages of processing before its amount is debited and credited to the proper accounts and the physical check is returned to its issuer. Take a look at a check that is returned from the bank; it has a number of rubber stamp marks on its back, perforations in it, and some additional MICR encoding (Magnetic Ink Character Recognition) on its face. All of that (except the MICR encoding) was at one time done by clerks, at great cost. It is all now done (again, with the exception of the MICR encoding of the amount) by machine, at the rate of millions of checks per hour.

2. Improved accuracy. This is fairly obvious. Those functions that are performed by machine—assuming that the machine is once programmed to do them correctly—will be done correctly over and over. People are highly error-prone; machines are reliable, and untiring. (Machines also do not require coffee breaks, and demand relatively little sick leave.)

3. Improved feedback. Consider a department store billing procedure that is done by hand (e.g., with bookkeeping machines and human operators), and that involves some form of time payments. Suppose that there is an error; for example, a payment is made to your account but an interest charge has been applied just before the payment was credited. You go to the store with all your papers to straighten out their thinking on your bill. You wait in line, and then explain the problem to a clerk. Your case is sound, and the clerk makes the proper adjustment to your account.

The next person in line has the same problem (with slight variations, perhaps). The process is repeated—and may be repeated hundreds of times. In fact, the people who do not take the trouble to stand in line and complain may be cheated. They may not even know that they are being cheated. Let us assume that it is an honest error and the department store is not deliberately cheating its customers.

On the other hand, if the billing procedure has been computerized, and an error in logic is uncovered, and that error is corrected, then we have something new in the world. The error *stays* corrected, and the correction is applied to everyone uniformly. Both the customers and the store have positive feedback on the logic of the system. This particular benefit of computerized paperwork systems probably concerns more people than the lowered costs and the increased accuracy. (This should not be taken to mean that this positive feedback comes automatically, or even most of the time. The *changes* must be made by people. They must be intelligent enough to recognize that a problem exists and that the cure is available, and they must have the authority and motivation to initiate the required change. These conditions are not commonly met. Nevertheless, the provision exists for mass-correction of errors, and it did not exist prior to computerized accounting.)

We could dwell on this potential for social benefit at great length. Suppose that it is decided, that as a social good, all juvenile offenses be expunged from police records when the person reaches age 18. With manual methods of information processing, there is simply no way to implement this notion completely and efficiently. A large police department may have 100,000 records on criminals, and a fairly large portion of these may contain juvenile offenses. It is too much to expect that these records will be cleaned up even once a year, to delete all information when a person reaches age 18. Moreover, it is only human nature to make use of information that is in a file, even if the decision was made not to count offenses that took place years ago. But if the file is computerized, and the decision is properly programmed, then it can be implemented easily and cheaply. What is even better, the purging of the file can take place daily, and without further human intervention. The juvenile record can simply vanish when the person reaches age 18.

Another example would be book club activities. We have had a rash of book and record clubs that systematically engaged in sending out unwanted and unordered books or records to their members. (That is, the members failed to send in the card saying "Do not send me anything this month"). The clubs then proceeded to bill the members endlessly, with interest charges added. Various court decisions decreed that this procedure was poor, and that the book clubs should improve their paperwork processing. As soon as the proper procedures could be programmed (and all the book clubs rely heavily on computers), millions of customers benefitted. The ability of the computer to alter a procedure en masse can operate to society's advantage.

To get an idea of how computers are applied to practical problems, we can consider the uses to which computers are put in the automobile industry.[1] The Ford Motor Company had, in 1973, 416 computers, which represented, in terms of their rental value, the fifth largest commercial user in the world. Many of these machines were occupied with production control for Ford. The company produces an average of over 10,000 cars and trucks each working day, with a peak of 17,000 in one day. This is done largely without warehouses; that is, all the parts and components of those 10,000 vehicles are scheduled to arrive at the assembly plants on the day they are needed. The assembly plants average 71,000 receiving transactions per day, where a "transaction" can be the arrival of a freight car full of batteries. The logistics problem in modern automobile assembly is staggering, and would be completely impossible without computer control. It involves hundreds of pipelines from all over the country (Ford has some 2000 suppliers) feeding the right parts to the assembly lines at precisely the right time. Getting those pipelines moving in synchronization is the largest problem, but there is added to that the once or twice yearly model changeover, when the entire apparatus must be throttled down and then restarted with new parts.

Computer control of such a complex operation is illustrative of the payoff that can result from intelligent application of computers, leading to reduction of inventory levels, reduction of bottleneck situations (which lead to expensive expediting, overtime, and premium freight payments), and lowering of costs due to obsolescence and leftover parts.

[1]The facts discussed here are taken from the article, "Information Processing in Manufacturing" by J. Paul Bergmoser, in the book *Manufacturing Management Systems*, edited by Fred Gruenberger, New York: Hayden Book Company, 1974.

PC60-16

6.3 MORE GOOD EFFECTS

1. *Traffic control.* If you drive a car in or near any large city, you should be painfully aware that traffic control could be improved. The objective should be to move cars, not impede them, and yet it usually appears as if traffic lights and other devices are installed for the sole purpose of minimizing traffic movement (see Figure 6.1).

What would it take to reverse this trend and build a traffic control system in which the movement of vehicles dictated the setting of the lights? First of all, there would have to be an information gathering function—a sensing mechanism—to determine, continuously and rapidly, where the cars are, where they are headed, and at what speed. Then there must be an information processing mechanism (i.e., computers) to analyze the situation and make the proper decisions. Finally, the results of those decisions must be fed back to the traffic lights. It is not a simple problem, and the solution is expensive. But the solution is feasible and costs less than the construction of new roads, off-street parking facilities, and the enforcement of traffic laws about one-way streets and the like. At the moment, the notion of computerized traffic control is still in the experimental stage, but in two cities where it has been tried—Toronto, Ontario and San Jose, California—it seems to be a qualified success.

Traffic control is one example of what is sometimes called real-time data processing (Figure 6.2), which means that the movement of information within the computer must closely synchronize with corresponding events in the real world. A more modern term is *on-line* data processing.

2. *Medical aids.* Consider the medical problem of interpreting an electrocardiogram (EKG, or ECG). An ECG trace is a long piece of paper on which voltage readings are recorded from several sensors attached to the patient. The shape of the wave forms tends to reveal, to the experienced diagnostician, something about the physical condition of the patient. The expert has examined many previous ECGs, both from people with normal heart action and from people known to have heart troubles.

Human analysis of ECG traces is far from perfect. There have been many instances in which the opinions of even groups of experts have been wrong, as evidenced by the subsequent condition of the patient. Nevertheless, the use of ECGs is a useful and common diagnostic tool. The trouble is the tool is awkward and expensive. ECGs are usually taken in a physician's office, by a trained technician, and the traces are analyzed by experts who can do only so many in a day; moreover, the supply of these experts is limited.

Whatever it is that the expert looks for on an ECG (distances between peaks on the curves; steepness of the wave fronts; correlations between the various curves, and so on—these are all problems in pattern recognition, which is a task that has long been done well only by people). Perhaps a computer program could be written to do the same thing. If so, and we could also solve the problem of how to "read" the ECG and translate its (analog) wave shapes into digitized information, then perhaps we could bring the cost of ECG analysis down sharply, and extend the technique to masses of people.

The computing problems involved in analyzing ECGs have all been solved and are in use at some university medical centers, but mass use is still pending.

3. *Jobs.* Prior to 1955, there were relatively few computers in operation, they were all expensive, and they were all devoted to mathematical problems. The total value of the computing hardware in the United States in 1955 was about $100 million.[2] The number of people who could call themselves part of the industry was 1000-2000.

From 1955 to 1974, some 85,000 machines were installed. Perhaps 30 percent of these machines were obtained to do mathematical, scientific, and engineering work, but the other 70 percent were devoted to data processing work; that is, file processing and record keeping—activities that affect you. The total value of these machines is measured in billions of dollars. Large numbers of people wholly dedicated to the computing industry are broken down roughly as follows:

IBM employees	250,000
Employees of all other computer vendors	250,000
Data processing personnel in other industries (including programmers)	500,000
Employees of ancillary companies (e.g., forms companies, peripheral products)	100,000
TOTAL	1,100,000

Here we have an industry that is two decades old, producing physical products valued at some $7.5 billion per year, and employing people with a combined salary of nearly $15 billion per year. All of this money is spent to perform work better, faster, cheaper, and more reliably than would otherwise be done by people. This implies that the use of computers has displaced people, which may be taken as a euphemism for "putting people out of work." Progress *always* displaces people. Automatic switching displaced telephone operators; pushbuttons and relays displaced elevator operators; signal lights displaced traffic police; bulldozers replaced men with picks and shovels. The fact that computers displace clerks is not something new. At a rough estimate, perhaps 10 million people have been displaced by computers. The computing industry itself has created new jobs for about 1 million people—and these are not the same people as those displaced. A clerk whose job has been eliminated does not turn into a computer designer, or a computer salesman, or a programmer.

But if 9 million people were put out of work, and no new jobs were created *for them* we would have had a serious social problem. In fact, when unemployment in the United States reaches a level of 5 million people (from all causes), it becomes a serious political issue as well as a grave social ill. If computers have been responsible for displacing millions of people, then at the same time millions of new jobs must have been created, in areas other than the computing industry itself. Evidence can be presented that some of these new jobs are due to the computer. In the following section, we will explore how this can come about.

[2] As has been pointed out earlier, the computing industry is noted for having few facts about itself. All figures in this section are invented, but they are probably correct within 20 percent.

6.4 LENS DESIGN

Let us follow through some of the problems involved in the design of a camera lens. Up to the early 1950s, each optical company employed an expert in this work; there were perhaps a dozen or so skilled lens designers in the world.

(We want to show how it is possible for the use of computers to *create* jobs by the thousands. The discussion that follows may be overly technical; the details may safely be skipped.)

When a new lens is needed, its design is subject to these constraints:

1. *Speed.* The light-gathering power of a lens is expressed as the ratio of the focal length to the aperture. Thus, an f2.0 lens has a focal length (the distance from the lens to the focal plane) twice as long as the lens opening at its widest.

2. *Cost.* A compound lens is made up of several elements, and for each of these elements a wide variety of glass types is available. The lens designer is limited by manufacturing cost considerations as to the number of elements he can use, and the rarity of the glass he requires.

3. *Quality.* Tolerance limits must be set on the acceptable level of the aberrations in the lens. Four main aberrations are:

(*a*) Spherical aberration: the degree to which the lens does not reproduce true circles as circles in the focal plane.

(*b*) Chromatic aberration: the degree to which a lens does not focus different colors of light the same.

(*c*) Astigmatism: the inability of a lens to focus horizontal and vertical lines the same.

(*d*) Coma: the tendency for a lens to transmit circles as amoeba-shaped figures.

The various aberrations fight each other; that is, a correction in the design for one of them tends to accentuate the others.

The lens designer manipulates the number of elements, the type of elements, the type of glass in each element, and the spacing of the elements in the lens tube.

He starts with a known lens (from a library of perhaps 25,000 lenses) that is close to the one desired. He then manipulates his four variables, and examines the result of his actions. This is done by ray tracing. Rays of light are traced through the lens surfaces by applying the standard ray tracing equations:

$$\sin I = \frac{L-r}{r} \sin U$$

$$\sin I' = \frac{n}{n'} \sin I$$

$$U' = U + I - I'$$

$$L' = r\frac{\sin I'}{\sin U'} + r$$

where n and n′ are the indices of refraction of the two media on either side of the given surface; I and I′ are angles between the incident and refracted rays and the normal to the surface; U and U′ are angles with the axis of the lens; and L and L′ are distances from the lens surface to the aimed object point and the true object point. These equations carry the light ray across one surface; the calculation must be extended across all the surfaces. All the calculations are usually carried out to 12 digit precision. Lens designers have found that they need to trace only 12 different rays through all the lens surfaces and interfaces to determine the characteristics of the proposed lens.

When such work was done by mechanical desk calculators it took up to two days to perform the ray tracing for one change in the design of a lens. The designer would then examine the results and, based on his experience, make some change in the variables and have the new design traced again. The lead time for a new lens (circa 1950) was about 2 years.

The ray tracing calculations are obviously amenable to computerization. A lens configuration can be traced in a second with little possibility of numeric errors. The designer could sit at the computer and manipulate the variables freely and easily, to reach his design goal quickly. Among other advantages, he would need to think about only one lens at a time, and work at that lens until he was satisfied.

The whole procedure can be further improved by writing a program to shift the variables automatically, according to criteria dictated by experience, to the point where the entire design process could take place without human intervention.

Where does all this take us? In a world that had just 12 lens designers, with a lead time of about 2 years for a new lens, high grade cameras were available only to professionals and rich amateurs. A camera with an f2.0 lens was rare and expensive. With the lens design problem automated by computer, anyone with access to a lens design program can design an excellent lens, and the lead time is measured in hours. It is possible to produce a camera with an f1.7 lens (that's 1.38 times as fast as an f2.0 lens) and sell it at the corner drug store to thousands of people. In a sense, 12 highly skilled people have been displaced, but new jobs have been created for many thousands of other people in the camera industry.

The lens design problem was described entirely in terms of camera lenses, because people normally associate cameras with lenses. But the lens industry is much larger than that. Lenses are vital to all sorts of devices such as telescopes, microscopes, copying machines, microfilming machines, and rangefinders. The capability, due to the computer, of designing new lenses has certainly opened up new markets and hence has created new jobs.

The typeset material above (less the half-tone Figures) is part of Chapter 6 of the book

<u>Computers and the Social Environment</u>

Copyright 1975 by John Wiley & Sons, Inc.
Published by Melville Publishing Company
Reprinted by permission of John Wiley & Sons, Inc.

PC60-20

Quality in computing science
is difficult to achieve
and teach:

when complexity is underestimated
"Anyone can do it easily"
when form, style, structure are scorned
"It's unnecessary and must be inefficient"
when flowcharts don't flow
"Sure they flow" ... all over.
when disposable parts are created
"We will use them once, then throw them away"
when testing, verification, or proof is not required
"It's too hard, and takes too long"
when errors (bugs) are expected
"It never works the first few times"
when cleverness is cherished
"That's too simple, stupid"
when optimism runs rampant
"I'm 95% finished" (for 95% of the time)
when disproportunate effort is spent on maintenance
"Too much time on design leaves insufficient for debugging"
when the ends justify any means
"It ran, see"
when documentation is non existent
"It understands me".

John Motil